Computer Science Department

TECHNICAL REPORT

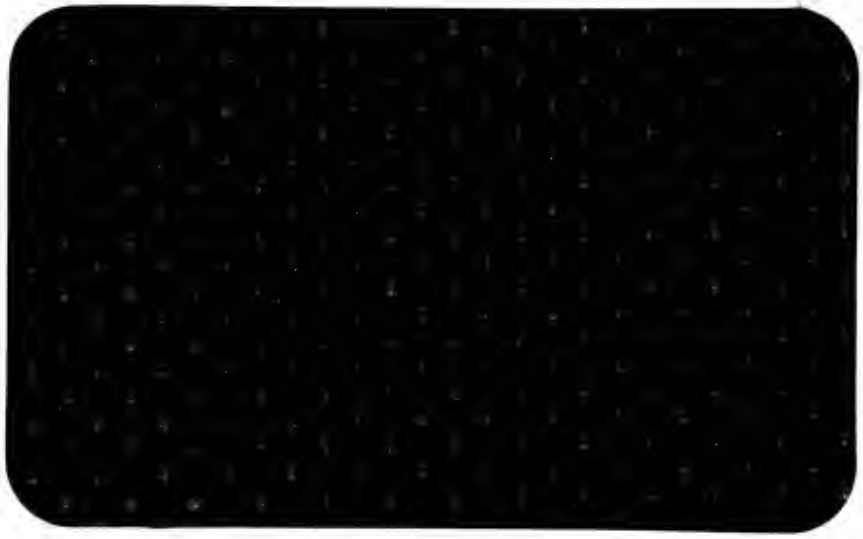
APPLICATION OF THE USE-DEFINITION CHAINING TO ATTRIBUTE-FLOW ANALYSIS *

by

Micha Sharir

January 1980

REPORT NO. 017



APPLICATION OF THE USE-DEFINITION CHAINING TO ATTRIBUTE-FLOW ANALYSIS *

by

Micha Sharir

January 1980

REPORT NO. 017

* This work is based upon work supported partly by the National Science Foundation under Grant Nos. MCS-7818922 and MCS 76-00116. Any opinions, findings and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

This work is also based upon work supported in part by the U.S. Department of Energy Contract EY-76-C-02-3077.

Table of Contents

ABSTRACT

Use-definition chaining is a well known pragmatic technique for solving various classes of data flow analysis problems. In this paper we compare this method with the standard flow graph method for solving such problems and show that when applied to the class of attribute flow analysis problems (which includes constant propagation, type analysis, etc.) both methods yield the same solution, but that the use definition method is generally much more efficient than the flow graph method. We also describe and analyze a possible adaptation of the use-definition technique to inter-procedural attribute flow analysis.

1. Introduction

Data flow analysis of computer programs has become a key tool for program analysis and optimization. However, a significant portion of the extensive literature on this subject concerns itself with abstract theory and methods, and the issue of practical implementation of these abstract methods does not always receive adequate treatment. As the role of code optimization tends to become more central in compiler design and implementation (especially in the development of progressively higher level programming languages), more thought will need to be given to the efficiency of existing data flow algorithms and methods.

This paper aims to narrow the gap between theory and practice by providing a theoretical study of the well known, but little analyzed data flow method of attribute propagation along definition-use links ([Al_1]; see below for detailed definitions). This method propagates data attributes from variable definitions directly to variable uses, rather than propagating these attributes along flow graph edges between basic blocks (for which approach see e.g. [He]). We will compare these two methods for an important class of attribute flow problems, in which attributes such as value, type and range are to be computed per each variable occurrence. We will show that both techniques yield the

same solution when applied to such an analysis problem, but that in general the use definition method is much more efficient and requires much less space and time than the flow graph method. Our analysis will allow us to relate various known flow graph based algorithms for performing analyses of this class to more efficient algorithms based on definition-use propagation. We note that several recent data flow algorithms (cf. e.g. [JM_1], [KaU], [JM_2], [Ka]) are usually so prohibitively expensive in their space (and time) requirements as to be impractical for use in a pragmatic general purpose optimizer unless they are reformulated in terms of the use-definition technique. Indeed, in reviewing recently published attribute flow analysis algorithms, one can usually distinguish easily between those that have been devised for pragmatic application (cf. [Te],[Ha]) and employ the use-definition technique, and those that have not (cf. [JM_1], [Ka], [KaU]).

Relevant notation and terminology will be introduced in section 2, in which we will describe the two methods mentioned above in some detail, including data flow frameworks to be used, data flow equations to be solved and the relation of their (maximal fixpoint) solution to the actual variable attributes to be computed. In section 3 we will prove equivalence between these two methods. Our main theorem (Theorem 3.5) shows that the application of the flow graph based method to an attribute flow problem, which yields attribute information

at entry to each basic block, and the local propagation of this information through the block assign to each variable occurrence an attribute identical to that that would have been obtained by applying the use-definition propagation technique. The proof is more complicated than might be expected, due to the fact that the data flow frameworks of most important attribute flow analyses are not distributive (cf. [He]). In the distributive case, a much shorter proof of this theorem which would also show that the two methods yield the 'meet over all paths' solution ([Ki], [He]) can be given. However, since for nondistributive frameworks this second solution is not recursively calculable, we must give a more complex proof, and also content ourselves with showing that both the flow graph and the use-definition techniques attain the same (possible) underestimate of the 'meet over all paths' solution.

Section 4 will then discuss the efficiency of the use-definition approach and will note several other pragmatically useful features of the use-definition map. Section 5 considers adaptation of the use-definition method to interprocedural analysis and discusses various problems arising in this case.

2. Terminology and Notation

This section introduces relevant notation and terminology. The reader is referred to [He] and [AU] for basic terminology concerning data-flow analysis.

The class of data flow analyses considered in this paper is the class of attribute-flow analyses, in which we wish to compute a certain attribute (value, type, range etc.) for each variable occurrence in a program P. This class of analyses includes constant propagation analysis [Ki], in which the required attribute is a value, type analysis ([Te],[JM_1],[KaU]), and range analysis [Ha].

A data flow framework for an attribute flow analysis of the class which interests us can be defined as follows: Let L_0 denote a lattice of attributes that can be acquired by program variables. Let Σ denote the set of all the variables occurring in the program P, which is to be analyzed. Then the lattice L used in our analysis is defined to be L_0^{Σ}, i.e. the set of all maps from Σ to L_0, with meet defined on Σ in a pointwise manner. We assume that L_0 is a well founded lattice. This makes L a well founded lattice, since Σ is always a finite set. Let $\Omega \in L_0$ be a largest element, indicating an undefined attribute (i.e. attribute of undefined (uninitialized) variables), and let $\Omega^* \in L$ denote the map which maps each $V \in \Sigma$ to Ω.

Example. In constant propagation, L_0 can be taken to be VALUES $\cup \{\Omega,\omega\}$, where VALUES is a set of relevant run time values that can be acquired by the variables in Σ, where ω is the smallest element of L_0 indicating more than one possible value, and where the meet of any two different elements of VALUES yields ω.

(Note that in this example, smaller elements of L_0 correspond to less specific attributes, and that meet in L_0 corresponds to taking the logical 'or' of the predicates on on program states describing the respective elements of L_0. This is compatible with standard data flow notation (as in [KU], [Ro_1] etc.)).

To define the set F of data propagation maps associated with L, we proceed as follows: With an instruction I having the form

$$V := \underline{op}\ (V_1,V_2,\ldots,V_k)$$

in the program to be analyzed we associate a 'shadow' instruction $\hat{I}$, describing the way in which the attribute of the output variable V depends on the attributes of the input arguments $V_1,\ldots,V_k$. This $\hat{I}$ will have the form

$$\alpha := A_I(\alpha_1,\alpha_2, \ldots, \alpha_k) ,$$

where $\alpha_1,\ldots,\alpha_k \in L_0$ are the assumed attributes of

$V_1,\ldots,V_k$ respectively, and $\alpha \in L_0$ is the resulting attribute of V. A_I is assumed to be a monotone map from L_0^k to L_0 (i.e. monotone in each of its arguments). We then associate a function $H_I: L \to L$ with $\hat{I}$ as follows: For each $x \in L$, $W \in \Sigma$, define

$$H_I(x)(W) = \begin{cases} A_I(x(V_1),\ldots,x(V_k)) & \text{if } W = V \\ x(w) & \text{if } W \neq V \end{cases}$$

(This formula propagates information through an instruction from input occurrences to output occurrences, and is therefore appropriate for data flow problems in which information is to be propagated in the direction of execution flow (called problems of the forward type), which are the only problems considered in this paper. However, problems involving propagation in the reverse direction of execution flow can be treated by methods like those described below, even though treatment of such backward problems usually requires construction of additional maps which modify the attributes of input variables as well).

Note that instructions with no output variables are transparent to the data flow propagation, and their H_I map can be taken to be the identity on L. Note finally that H_I is monotone in L since A_I is monotone in L_0.

We then define F as the smallest set of functions acting on L which is closed under functional compositions

and meets and contains the identity map and all the above functions H_I.

Having constructed this framework, we can apply the standard flow graph data flow analysis technique to it. This technique assumes that the program being analyzed is represented by a flow graph G, which is a rooted directed graph whose set of nodes N is the set of all program basic blocks (i.e. single entry code sequences), whose edges are of the form (m,n), where m,n are basic blocks such that n can be executed immediately after m, and whose root (entry node) is the entry block r of the program (i.e. the block at which execution starts). For simplicity, that is assumed not to belong to any cycle in G. In this initial model we ignore interprocedural transfer of control (which will be discussed in section 5) and thus assume the program being analyzed to contain no subprocedures. To simplify our analysis, we will assume that basic blocks are also single exit and that the last instruction in each block is a branch instruction.

We can then associate with each edge $(m,n) \in G$, a data flow map $f_{(m,n)}$ defined as follows: Let m consist of the instructions $I_1, I_2, \ldots, I_j$ in order, and put

$$f_{(m,n)} = H_{I_j} \circ H_{I_{j-1}} \circ \cdots \circ H_{I_1}$$

This shows the effect on L of the advance of control from the start of m to the start of n. (Note that in our model

$f_{(m,n)}$ is independent of n, and that the last (branch) instruction has really no effect on the data flow). Obviously, $f_{(m,n)} \in F$.

Having defined these maps, we look for the (maximal fixpoint) solution of the following standard data flow equations (where $x_n \in L$ denotes attribute data to be computed at the start of the basic block $n \in N$):

$$\begin{aligned} x_r &= \Omega^* \\ x_n &= \Lambda\{f_{(m,n)}(x_m) : (m,n) \in G\}, \quad n \in N-\{r\} \end{aligned} \tag{2.1}$$

It is well known ([Ki]) that a maximal fixpoint of these equations exists, and can be found by successive approximation using a variety of iterative techniques, such as 'workpile propagation' [Ki] or round-robin iteration through the nodes of N [HU]. Note that the solution of (2.1) only yields information at basic block entries, and an additional step of propagation through each basic block is required. Specifically, for a block $n \in N$ consisting of the sequence $I_1,\ldots,I_j$ of instructions, and for each $k \leq j$ we compute w_{I_k}, the attribute data state at the start of I_k, using the formula

$$w_{I_k} = H_{I_{k-1}} \circ \cdots \circ H_{I_1}(x_n) \tag{2.2}$$

Then for each input argument V of I_k, $w_{I_k}(V)$ yields the

attribute (in L_0) ascribed to V immediately prior to execution of I_k . If V is an output variable of I_k , then its attribute (known just after the execution of I_k) is $H_{I_k}(w_{I_k})(V)$.

It is well known that the maximal fixpoint of (2.1) need not coincide with the more accurate 'meet over all paths' solution (see [He] where this is demonstrated for constant propagation analysis). Nevertheless, our solution is always safe, in the sense that the attributes that we compute are always a lower bound in L_0 to the attributes that can be actually acquired at run time at the respective variable occurrences. (See [Ro_1] for a more general discussion of these issues.)

Attribute flow analysis can be also accomplished using a second more efficient technique, which we will call the use-definition propagation. This approach begins by computing a use-definition chaining map 'ud', which is defined as follows [Al_1]: for each use (input occurrence) VO_1 of a variable $V \in \Sigma$, define $ud\{VO_1\}$ to be the set of all definitions or modifications (output occurrences) of V from which VO_1 can be reached along an execution path which is free of any other definitions or modifications of V. This map can be computed by performing a standard reaching definitions data-flow analysis ([He], [AU]). Since this analysis involves a very simple framework, it can use any one of several efficient data flow algorithms ([HU], [AC], [GW], etc.).

Once having computed the ud map, we can perform attribute flow analysis using the following ud propagation technique: Let 'attr' denote a map sending each variable occurrence in the program being analyzed to its ascribed attribute (in L_0). This map should satisfy the following set of equations

$$\text{(i)}\quad attr(VO) = \bigwedge\{attr(VO') : VO' \in ud\{VO\}\}$$
$$\text{for each variable } \underline{\text{use}} \text{ VO}$$

$$\text{(ii)}\quad attr(VO) = A_I(attr(VO_1),\ldots,attr(VO_k)) \tag{2.3}$$

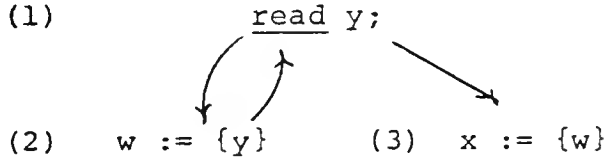
for each variable modification (definition) VO appearing as an output variable of an instruction I, having $VO_1,\ldots,VO_k$ as input variable occurrences

Equations (2.3) can be solved iteratively by successive approximation to obtain a maximal fixpoint solution, which exists since these equations are monotone in 'attr', viewed as an element of the (well founded) lattice of maps from variable occurrences into L_0.

To solve these equations, one can use either the workpile propagation technique of [Ki], or a round-robin iteration as in [HU], with the 'attr' map initially set to map all variable occurrences to Ω. Note that a program will contain variable defining instructions having no input (variable) arguments, such as <u>read</u> instructions or operations on constants. The A_I function corresponding to such an

instruction I will then be constant, so that the attribute of the output occurrence of I will be constant independent of any attribute propagation. If the workpile propagation solution technique is used, then propagation should start at the output occurrences of all these instructions; if a round robin iteration is used, no special treatment need be given to these instructions.

Remark. Equations (2.3) (i) reflect the assumption that the attribute of a variable V is not changed between a point at which V is defined and a subsequent point at which V is used, so that the attribute computed for the definition of V can be propagated directly to the use. This assumption will fail for some relational attribute flow analyses, in which the attributes being sought are actually relations between variables. A typical example of this sort is found in Kaplan [Ka, p. 14], who discusses value flow analysis (cf. [Sc_1]). This example contains roughly the following code:



Suppose here that the attribute map that we wish to compute is 'crpart' which maps each variable occurrence VO to the set of all prior output occurrences whose value might become

part of the value of VO. Such a relational attribute map need not satisfy the above assumption. Indeed in this example there exists a path (2,1,3) from the definition of W at (2) to its use at (3), such that the 'crpart' attribute of W changes along this path, because

$$\text{crpart}(W \text{ at } (2)) = \{y \text{ at } (1)\},$$

but

$$\text{crpart}(W \text{ at } (3)) = \emptyset$$

In what follows, we will restrict our discussion to attribute flow analyses in which variable attributes do not change unless the variable is modified. It is interesting to note that this difficulty is missed in [Sc_1].

A similar situation can also arise in cases where program variables can be aliased to each other, either by passing procedure parameters by reference, or by using pointer variables. In such cases a variable attribute may change due to a modification of an aliased variable. We will therefore also assume that no variable aliasing takes place in the program being analyzed. When such aliasing does occur, the use-definition propagation technique must be used with caution, modifying it to take into account all possible aliases of variables, and even then it may fail to produce as sharp a solution as the flow-graph technique.

3. Equivalence of the flow graph and the ud propagation methods

In this section we show that the two attribute propagation techniques described in section 2 yield identical solutions when applied to attribute flow analyses for which our assumption concerning attribute preservation between variable definitions and subsequent uses is valid.

To facilitate the proof of this claim, we first convert the flow graph approach into another approach, somewhat closer to the ud propagation technique, as follows: Rather than partitioning the given program into basic blocks, we partition it into single instruction blocks to obtain a new and larger flow graph G_I whose set of nodes N_I consists of all program instructions, whose entry node I_0 is the entry instruction of the program (or procedure), and whose edges are of the form (I,J) where I and J are control-adjacent instructions (that is, are instructions for which control can be transferred directly from I to J).

We can then use the data flow framework (L,F) in an analysis of the new flow graph. This calls for the solution of the following attribute flow equations (where, for each $I \in N_I$ we let $z_I \in L$ denote the attribure map attached to the program point which immediately precedes I):

$$z_{I_0} = \Omega^* \in L \ ,$$

(3.1)

$$z_I = \{H_J(z_J): (J,I) \in G_I\} \quad \text{for each } I \neq I_0 \ .$$

(Note that H_J represents the change in attributes which takes place when control passes from J to any of its successors; this change depends solely on J itself.) These equations can be solved iteratively by any of the standard propagation techniques mentioned in section 2, setting initially $z_I = \Omega^*$ for each $I \in N_I$.

As a first step in our analysis, we show the following

Theorem 3.1. For each $n \in N$, $x_n = z_{I_n}$, where I_n denotes the initial instruction of n.

Proof: Let $n \in N$ be the sequence $J_1,\ldots,J_k$ of instructions. As in (2.2), define

$$w_{J_t} = H_{J_{t-1}} \circ \cdots \circ H_{J_n}(x_n) \tag{3.2}$$

for each $t \leq k$. We claim that $\{w_J\}_{J \in N_I}$ is a solution of Equations (3.1). Obviously, $w_{I_0} = x_r = \Omega^*$. Let $I \in N_I$ be an instruction which is not the initial instruction in its basic block. Then I has a single predecessor J, and (3.2) implies that $w_I = H_J(w_J)$, so that (3.1) is satisfied in this case. If $I = I_n$ for some basic block $n \neq r \in N$ then, by (2.1),

$$w_I = x_n = \Lambda \left\{f_{(m,n)}(x_m): (m,n) \in G\right\}$$

But for each such m, $f_{(m,n)}(x_m) = H_J(w_J)$, where J is the branch from m to n. Hence

$$w_I = \Lambda \left\{H_J(w_J): (J,I) \in G_I\right\}$$

(Note that these J are precisely all the predecessors of I in G_I.) Therefore (3.1) is satisfied by the w_J's, and since the z_j's are the maximal fixpoint of (3.1) we must have

$$w_J \leq z_J \text{ , for each } J \in N_I \text{ .}$$

Conversely, for each $n \in N$, define $\hat{x}_n = z_{I_n}$. We claim that $\{\hat{x}_n\}_{n\in N}$ satisfies Equations (2.1). Indeed, $\hat{x}_r = z_{I_0} = \Omega^*$. Let $n \neq r \in N$. By (3.1) we have

$$\hat{x}_n = z_{I_n} = \Lambda \left\{H_J(z_J): (J,I_n) \in G_I\right\}$$

But an instruction J appears in the above meet iff there exists $m \in N$ such that $(m,n) \in G$ and J is a branch instruction from m to n (i.e. to I_n). Hence

$$H_J(z_J) = f_{(m,n)}(z_{I_m}) = f_{(m,n)}(\hat{x}_m)$$

so that

$$\hat{x}_n = \Lambda \left\{f_{(m,n)}(\hat{x}_m): (m,n) \in G\right\}$$

which establishes our claim. Hence $z_{I_n} = \hat{x}_n \leq x_n$ for each $n \in N$ which implies the assertion of our theorem. Q.E.D.

Next, we utilize the special structure of the functions $\{H_I\}_{I \in N_I}$ to prove the following.

Lemma 3.2. For each $I \in N_I$ and $V \in \Sigma$ we have

(3.3) $z_I(V) = \wedge \{H_J(z_J)(V)$: V is modified in J and there exists a path free of modifications of V from J to I$\}$

Proof: Denote the right-hand side of (3.3) by $\hat{z}_I(v)$. Let $I \in N_I$ and $V \in \Sigma$ be given, let $J \in N_I$ be an instruction appearing in the meet in (3.3) and let $(J = K_1, K_2, \ldots, K_s = I)$ be a path free of modifications of V. Then, by (3.1) and the definition of the functions H_K, we have

$$\begin{aligned} z_{K_2}(V) &\leq H_J(z_J)(V) \\ z_{K_3}(V) &\leq H_{K_2}(z_{K_2})(V) = z_{K_2}(V) \\ &\vdots \\ z_I(V) &\leq H_{K_{s-1}}(z_{K_{s-1}})(V) = z_{K_{s-1}}(V) \end{aligned}$$

Hence $z_I(V) \leq H_J(z_J)(V)$, so that $z_I(V) \leq \hat{z}_I(V)$.

Conversely, assume that equations (3.1) are solved using a workpile-driven propagation scheme, as in [Ki]. For each $k \geq 0$ let z_I^k denote the value of z_I after k propagation steps of this iterative process, and let $\hat{z}_I^k$ denote the corresponding value of $\hat{z}_I$. Then we claim that

(3.4) $$z_I^k(V) \geq \hat{z}_I^k(V) \quad , \quad I \in N_I \ , \ V \in \Sigma \ , \ k \geq 0 \ .$$

The proof is by induction on k. Initially we have $z_I^0(V) = \Omega \geq \hat{z}_I^0(V)$ for each I, V. Suppose that (3.4) is true for some $k \geq 0$, and consider the (k+1)-st propagation step which propagates data from some instruction $\tilde{J}$ to one of its successors $\tilde{I}$. Let $I \in N_I$, $V \in \Sigma$. If $I \neq \tilde{I}$ then, by the induction hypothesis,

$$z_I^{k+1}(V) = z_I^k(V) \geq \hat{z}_I^k(V) \geq \hat{z}_I^{k+1}(V)$$

If $I = \tilde{I}$ then

$$z_{\tilde{I}}^{k+1}(V) = z_{\tilde{I}}^k(V) \wedge H_{\tilde{J}}(z_{\tilde{J}}^k)(V)$$

By the induction hypothesis we have $z_{\tilde{I}}^k(V) \geq \hat{z}_{\tilde{I}}^k(V) \geq \hat{z}_{\tilde{I}}^{k+1}(V)$. If V is modified in $\tilde{J}$ then $\tilde{J}$ appears in the meet for the expression $\hat{z}_{\tilde{I}}(V)$. Since $z_{\tilde{J}}^k \geq z_{\tilde{J}}^{k+1}$ and $H_{\tilde{J}}$ is monotone in L, we obtain

$$H_{\tilde{J}}(z_{\tilde{J}}^k)(V) \geq H_{\tilde{J}}(z_{\tilde{J}}^{k+1})(V) \geq \hat{z}_{\tilde{I}}^{k+1}(V)$$

so that $z_{\tilde{I}}^{k+1}(V) \geq \hat{z}_{\tilde{I}}^{k+1}(V)$ in this case. On the other hand, if V is not modified in $\tilde{J}$, then

$$H_{\tilde{J}}(z_{\tilde{J}}^k)(V) = z_{\tilde{J}}^k(V) \geq \hat{z}_{\tilde{J}}^k(V) \geq \hat{z}_{\tilde{J}}^{k+1}(V)$$

But each instruction K appearing in the meet for the expression $\hat{z}_{\tilde{J}}^{k+1}(V)$ must also appear in the expression $\hat{z}_{\tilde{I}}^{k+1}(V)$, because the existence of a modification-free path from K to $\tilde{J}$ implies the existence of a similar path from K to $\tilde{I}$, obtained

by concatenating the edge $(\tilde{J},\tilde{I})$ to the first path. Hence we have $\hat{z}^{k+1}_{\tilde{J}}(V) \geq \hat{z}^{k+1}_{\tilde{I}}(V)$, so that in this case too we have

$$z^{k+1}_{\tilde{I}}(V) \geq \hat{z}^{k+1}_{\tilde{I}}(V)$$

This establishes (3.4), from which the lemma follows readily.

Q.E.D.

The result established in Lemma 3.2 can be expressed in a form showing the close connection between the 'attr' map of (2.3) and the map z, as follows. Introduce an auxiliary map $\widehat{\text{'attr'}}$, defined as follows: for each $I \in N_I$, $V \in \Sigma$ which is an input argument of I, put

$$\widehat{\text{attr}}(V_I) = z_I(V) \ ,$$

(where V_I denotes that occurrence of V at I); if V is an output variable of I, then define

$$\widehat{\text{attr}}(V_I) = H_I(z_I)(V) \ .$$

Corollary 3.3. For each $I \in N_I$ and each variable occurrence V_I in I we have

$$\widehat{\text{attr}}(V_I) \leq \text{attr}(V_I) \ .$$

Proof. We will show that the map '$\widehat{\text{attr}}$' satisfies Equations (2.3). Since 'attr' is the maximal fixpoint of these equations this will imply the above inequality. Suppose first that V_I is an input occurrence in I. By (3.3) and the definition of the ud-map we can write

$$\widehat{\text{attr}}(V_I) = z_I(V) = \bigwedge \left\{H_J(z_J)(V) : V_J \in \text{ud}\{V_I\}\right\}$$
$$= \bigwedge \left\{\widehat{\text{attr}}(V_J) : V_J \in \text{ud}\{V_I\}\right\}$$

which is precisely (2.3)(i). If V_I is an output occurrence in I, then

$$\widehat{\text{attr}}(V_I) = H_I(z_I)(V) = A_I(z_I(V_1),\ldots,z_I(V_k))$$

(where $V_1,\ldots,V_k$ are the input arguments of I),

$$= A_I(\widehat{\text{attr}}(V_{1,I}),\ldots,\widehat{\text{attr}}(V_{k,I}))\ ,$$

which is precisely (2.3)(ii). This proves our assertion.

Q.E.D.

Next we show that the two maps 'attr' and '$\widehat{\text{attr}}$' are identical.

Lemma 3.4. For each $I \in N_I$ and each variable occurrence V_I in I we have

$$\text{attr}(V_I) = \widehat{\text{attr}}(V_I)\ .$$

Proof. By virtue of Corollary 3.3 it is sufficient to prove that $attr(V_I) \leq \widehat{attr}(V_I)$ for each occurrence V_I. To do this, let $\widehat{attr}^k(V_I)$, $k \geq 0$, denote the value of this map after the first k propagation steps in the iterative solution of Equations (3.1), as in the proof of Lemma 3.2. We claim that

$$\widehat{attr}^k(V_I) \geq attr(V_I)$$

for each variable occurrence V_I and $k \geq 0$. To prove this, we use induction on k. If k = 0 then clearly

$$\widehat{attr}^0(V_I) = \Omega \geq attr(V_I) \ , \quad \text{for each} \quad V_I \ .$$

Suppose that these inequalities hold for some $k \geq 0$, consider the (k+1)-st propagation step, and suppose that this step propagates data from some instruction $\tilde{J} \in N_I$ to one of its successors $\tilde{I}$. Let $I \in N_I$. If $I \neq \tilde{I}$ then by the induction hypothesis,

$$\widehat{attr}^{k+1}(V_I) = \widehat{attr}^k(V_I) \geq attr(V_I) \ , \quad \text{for each } V_I \text{ in } I.$$

If $I = \tilde{I}$ and $V_{\tilde{I}}$ is an input occurrence in $\tilde{I}$ then

$$\widehat{attr}^{k+1}(V_{\tilde{I}}) = z_{\tilde{I}}^{k+1}(V) = z_{\tilde{I}}^{k}(V) \wedge H_{\tilde{J}}(z_{\tilde{J}}^{k})(V)$$

But $z_{\tilde{I}}^{k}(V) = \widehat{attr}^k(V_{\tilde{I}}) \geq attr(V_{\tilde{I}})$, by the induction hypothesis. If V is modified in $\tilde{J}$, then $H_{\tilde{J}}(z_{\tilde{J}}^{k})(V) = attr^k(V_{\tilde{J}})$ and $V_{\tilde{J}} \in ud\{V_{\tilde{I}}\}$. Hence, by (2.3)(i) and the induction hypothesis,

$$H_{\tilde{J}}(z_{\tilde{J}}^{k})(V) \geq attr(V_{\tilde{J}}) \geq attr(V_{\tilde{I}})$$

so that in this case $\widehat{attr}^{k+1}(V_{\tilde{I}}) \geq attr(V_{\tilde{I}})$. On the other hand, if V is not modified in $\tilde{J}$, then by (3.4) and its proof,

$$H_{\tilde{J}}(z^k_{\tilde{J}})(V) = z^k_{\tilde{J}}(V) \geq \hat{z}^k_{\tilde{J}}(V) \geq \hat{z}^k_I(V)$$

As in the proof of Corollary 3.3 we can then write

$$\hat{z}^k_{\tilde{I}}(V) = \Lambda \left\{ \widehat{attr}^k(V_J) : V_J \in ud\{V_I\} \right\}$$

$$\geq \Lambda \left\{ attr(V_J) : V_J \in ud\{V_{\tilde{I}}\} \right\} = attr(V_{\tilde{I}}) .$$

Finally, if $V_{\tilde{I}}$ is an output occurrence in $\tilde{I}$, then

$$\widehat{attr}^{k+1}(V_{\tilde{I}}) = H_{\tilde{I}}(z^{k+1}_{\tilde{I}}(V)) = A_{\tilde{I}}(z^{k+1}_{\tilde{I}}(V_1),\ldots,z^{k+1}_{\tilde{I}}(V_\ell))$$

where $V_1,\ldots,V_\ell$ are the input arguments of $\tilde{I}$. But by what we have just shown and by the monotonicity of $A_{\tilde{I}}$, we obtain

$$\widehat{attr}^{k+1}(V_{\tilde{I}}) \geq A_{\tilde{I}}(attr(V_{1,\tilde{I}}),\ldots,attr(V_{\ell,\tilde{I}})) = attr(V_{\tilde{I}})$$

by (2.3)(ii). This establishes our claim and completes the proof of our lemma. Q.E.D.

The following theorem now establishes the equivalence of the flow graph and the ud propagation methods.

Theorem 3.5. For each $I \in N_I$ let $w_I \in L$ be defined as in (2.2). Then

(i) If V_I is an input occurrence in I then $attr(V_I) = w_I(V)$

(ii) If V_I is an output occurrence in I then $attr(V_I) = H_I(w_I)(V)$.

Proof: Immediate from Theorem 3.1 and Lemma 3.4. Q.E.D.

Remark. A technique similar to ud-propagation, propagates data between variable occurrences linked by a modified data flow map 'udu'. This map links each use of a variable x to all prior modifications and uses of x, from which the given use can be reached along a path free of any other occurrences of x. As noted in [Sc_2], this approach is likely to be more efficient than the ud approach for typical programs. The same arguments used to prove Theorem 3.5 can be used to show that this modified technique also produces the same output as the flow graph method.

4. Various properties of the ud map

In this section we comment briefly on various pragmatically significant properties of the ud map. We have already shown that it can be used to perform attribute-flow analysis by direct propagation from variable definitions to their uses, and that this method yields the same results as the standard flow graph based method. To convince ourselves that use-definition propagation is much more efficient than flow graph propagation for attribute flow analysis, we note that the space required by the flow graph technique is $O(|N| \times |\Sigma|)$, since we have to compute an attribute map in L (which map will require space $O(|\Sigma|)$) for each node $n \in N$, even though most of the variables of Σ will not have appeared in n. The space required by the ud propagation method, ignoring the space needed to store the ud map itself, is linear in the number of variable occurrences in the program and for typical programs is therefore much smaller than the space required by the flow graph method. In fact, this space requirement is obviously the smallest possible for such a type of data flow analysis. Also, the space required for the ud map will usually be linear in the number of variable occurrences because in typical programs the number of definitions that can reach a given use is often very small (one or two on the average); and once computed, this map can be used in all subsequent attribute flow analyses. Once having computed

the ud map, we can in fact ignore the particular representation of the program to be analyzed, and perform attribute flow analysis by ud propagation, which is essentially independent of the program representation. Thus, the ud propagation method can also be used in cases where the program to be analyzed is represented by its parse tree or by other alternative representations, as in Rosen's high-level data flow analysis ([Ro_1]; cf. also [MFS]).

An additional disadvantage of the flow graph method is that the functions $f_{(m,n)}$ which it needs are functional compositions of rather complex functions, whose computation and storage may be infeasible in practice.

For all these reasons the ud propagation technique is preferable to the flow graph technique in pragmatic solution of attribute flow problems. Note however that the worst case behavior of the ud propagation method may be as bad as the behavior of the flow graph method, and even worse, due to the overhead of the preliminary calculation of 'ud'; but worst case programs are very unlikely to occur in practice.)

5. Issues in interprocedural ud propagation

The presence of procedures and procedure calls in the program to be analyzed creates special problems for any data flow analysis technique (cf. [Al_2], [Ro_2], [Ba], [SP]). The problem for the use-definition propagation method is twofold: (a) how to compute the ud map correctly in the presence of interprocedural flow, and (b) how to apply it in attribute flow analysis to obtain sharp information.

The first problem has by now been studied fairly extensively and several methods which compute accurate versions of the ud map interprocedurally are known (cf. [Al_2] for nonrecursive programs, [SS] for recursive programs with no variable aliasing; for cases where variable aliasing can occur, one can compute the ud map by combining an aliasing analysis such as in [Ba] with the interprocedural analysis technique of [SS], but as noted in Section 2, we will exclude this latter case from our study). Note that the main issue here is to establish correct links between global variable occurrences across procedures.

The second problem mentioned above has been less studied. To see what the problem is, consider the following example (where x, y are global variables):

```
        procedure P1                              procedure P2
           :                  procedure Q             :
(1)     x := 1                    :          (5)  x := 2
           :              (4)  y := x                :
(2)     call Q                    :          (6)  call Q
           :                    end;                 :
(3)     z := y                                  (7)  z := y
```

Any valid computation of the ud map will satisfy

$$ud\{x_4\} = \{x_1, x_5\}$$
$$ud\{y_3\} = \{y_4\}$$
$$ud\{y_7\} = \{y_4\} \quad .$$

Thus, in applying Equations (2.3) to some attribute flow analysis (say, constant propagation) we might propagate attributes from x_1 to x_4 , then to y_4 , then to y_7 and finally to z_7 , tracing an obviously spurious path, to conclude that z_7 has no constant value, although as a matter of fact it has the constant value 2.

This example shows that even if interprocedural ud links are computed accurately, their use in ud propagation solution of an attribute flow analysis may lead to inaccurate (but nevertheless overestimated, and hence safe) solution. The reason is that each ud link corresponds to some execution path (or set of such paths) along which this link can be realized. Thus attribute propagation along a series of such links, using Equations (2.3), amounts to tracing execution flow along the concatenation of such paths which, in the interprocedural case, may result in an interprocedurally invalid path.

The simplist approach to this problem is to ingore it, use Equations (2.3) as they stand, and obtain overestimated attribute information in situations as in the example given above. However sharper information can be obtained at the expense of more complicated algorithms.

In what follows we will suggest a possible approach based on the 'call-string' technique of [SP]. As will be seen below this approach is fairly complicated, but nevertheless interesting from a theoretical point of view. It can also be generalized to other situations where execution paths can be concatenated only in a selective manner, using the qualified data-flow analysis approach of Holley and Rosen [HR]. As indicated by their experience such an approach may still be pragmatically feasible in spite of its complexity, especially in cases where sharper information is desired and is likely to have a high payoff in the optimization of the program. (This is the case e.g. when the effects of in-line procedure integration are to be analyzed before actual expansion of the procedures.)

We will assume familiarity with the 'call-string' approach as outlined in [SP, Sections 4 - 6]. Its main advantage is that it constructs a modified data-flow framework (L*,F*) and uses it instead of a standard framework, (L,F) to set down data-flow equations which define the required solution in precisely the same way as is done in a purely intraprocedural data-flow analysis. As was done in [SP], we will first describe a purely abstract interprocedural use-definition approach, which in the general (recursive) case does not yield a finitely converging algorithm, but will serve to set down a framework in which a certain interprocedural version of the use-definition chaining map

can be defined and used in a manner rather analogous to the intraprocedural ud-approach. This framework with only minor modifications will then be used to obtain an effective, though approximative, interprocedural ud-technique for solving attribute flow analyses.

We start with a definition of an interprocedural version of the ud map. Most of our notations are taken from [SP]. In addition, we denote by Π the set of all variable occurrences in a program.

Definition. We define an *interprocedural use-definition chaining map*, denoted by ud*, as a relation in $\Pi \times \Gamma$ (Γ being the set of all call-strings), such that ud*{(VO,γ)} contains (VO_1,γ_1) iff VO is a use of some variable $V \in \Sigma$, VO_1 is a definition of V, and there exist execution paths $p_1 \in IVP(r_1,VO_1)$, $p_0 \in path_{G^*}(VO_1,VO)$ such that $CM(p_1) = \gamma_1$, $p \equiv p_1 \| p_0 \in IVP(r_1,VO)$, $CM(p) = \gamma$ and p_0 contains no definitions (modifications) of V. (Here we use a convention of not distinguishing between execution paths, leading from one instruction to another, and graph paths, leading from one basic block to another. In fact, the notions of interprocedural validity and CM value can easily be extended to execution paths.) Heuristically, we link each variable use VO and interprocedural 'flow-summary' γ tagging some execution paths leading to VO to a preceding definition VO' and flow summary γ' tagging prefixes of these paths. This will enable us to filter out invalid concatenations of these paths.

We omit here any details on computation of the ud* map which may not be effective if Γ is infinite (i.e. if the program to be analyzed is recursive). We will however explore this issue in detail later on.

As an example, consider the program given at the beginning of this section, and assume that both procedures P_1 and P_2 are called from a main program by the call instructions c', c" respectively. Then we have

$$ud^*\{(x_4,(c'2))\} = \{(x_1,(c'))\}$$

where (c'), (c'2) are call strings denoting the sequences of pending calls by which x_1 and x_4 have been reached. Likewise,

$$ud^*\{(x_4,(c"6))\} = \{(x_5,(c"))\}$$
$$ud^*\{(y_3,(c'))\} = \{(y_4,(c'2))\}$$
$$ud^*\{(y_7,(c"))\} = \{(y_4,(c"6))\}$$

An abstract flow graph approach in the interprocedural case which uses call strings is described in detail in Section 4 of [SP]. To obtain a corresponding ud^*-approach for attribute flow analyses, our objective is to compute a map $attr^*: \Pi \times \Gamma \to L_0$, which we shall represent as a map from Π into $L_0^* \equiv L_0^\Gamma$, such that for each $VO \in \Pi$, $\gamma \in \Gamma$, $attr^*(VO)(\gamma)$ is the attribute that VO attains if execution proceeds along paths in $CM^{-1}\{\gamma\}$.

The attr* map should satisfy the following set of equations (similar to equations (2.3)):

(5.1)

(i) $attr^*(VO)(\gamma) = \Lambda\{attr^*(VO')(\gamma') : (VO',\gamma') \in ud^*\{(VO,\gamma)\}\}$

for each variable use $VO \in \Pi$ and $\gamma \in \Gamma$.

(ii) $attr^*(VO)(\gamma) = A_I(attr^*(VO_1)(\gamma),\ldots,attr^*(VO_k)(\gamma))$

for each variable definition/modification $VO \in \Pi$ appearing as an output occurrence in some instruction whose input occurrences are $VO_1,\ldots,VO_k$, and each $\gamma \in \Gamma$.

Initialization of an iterative process to solve these equations (if a workset-oriented propagation is used) is carried out in a manner analogous to that mentioned in Section 2, i.e. by initializing attr* for all output occurrences VO of inputless instructions by their constant attributes. In doing so, we can either assign the corresponding attribute of VO to $attr^*(VO)(\gamma)$ only for $\gamma \in \Gamma$ for which $CM^{-1}\{\gamma\} \cap IVP(r_1,VO) \neq \emptyset$ (i.e. for which there exists an interprocedurally valid execution path from the program's entry to VO which is 'tagged' by γ), or else rely on the definition of the ud* map to ensure that no propagation that uses $attr^*(VO,\gamma)$ for γ's not satisfying the above condition will ever take place, as $ud^{*-1}\{(VO,\gamma)\}$ should be $\emptyset$ for such γ's.

Applying equations (5.1) to the example program considered above, it can easily be checked that one obtains

$$attr^*(x_4,(c'2)) = \text{constant value } 1$$

$$attr^*(x_4,(c"6)) = \text{constant value } 2$$

$$attr^*(y_4,(c'2)) = 1$$

$$attr^*(y_4,(c"6)) = 2$$

$$attr^*(y_3,(c')) = attr^*(z_3,(c')) = 1$$

$$attr^*(y_7,(c")) = attr^*(z_7,(c")) = 2$$

so that this technique enables us to deduce accurately e.g. that z_7 has the constant value 2, a fact which would have been lost by using the standard ud-propagation technique.

Having demonstrated the usefulness of the more sophisticated interprocedural ud* approach, we now proceed to describe and analyze this approach in greater detail.

As noted above, using the approach outlined in Section 4 of [SP] will yield an interprocedural flow-graph approach to attribute-flow analysis, which uses instead of the framework (L,F) defined in Section 2, a modified framework (L^*,F^*), where $L^* = L^{\Gamma} \cong (L_0^*)^{\Sigma}$, and F^* is defined as follows. Each instruction I which is neither a call nor a return instruction induces a function $H_I^* \in F^*$, so that for each $x^* \in L^*$, $\gamma \in \Gamma$

$$H_I^*(x^*)(\gamma) = H_I(x^*(\gamma))$$

where H_I is as defined in Section 2. If I is a call instruction (thus, as assumed in [SP], constituting a call block $m_I \in N^*$, calling some procedure p) then we define

$$H_I^*(x^*)(\gamma) = \begin{cases} x^*(\gamma_1), \text{ if there exists (necessarily a unique) } \gamma_1 \\ \qquad\qquad \text{such that } \gamma = \gamma_1 \circ (m_I, r_p); \\ \text{undefined, otherwise .} \end{cases}$$

If I is a return instruction from some procedure p (thus constituting the exit block e_p of p), then we define a separate function $H^*_{(I,J)}$ for each J that is an initial instruction of a block n_J which follows immediately a call to p, so that

$$H^*_{(I,J)}(x^*)(\gamma) = \begin{cases} x^*(\gamma_1), \text{ if there exists (necessarily a unique)} \\ \qquad \gamma_1 \text{ such that } \gamma = \gamma_1 \circ (e_p, n_J) \text{ ;} \\ \text{undefined, otherwise .} \end{cases}$$

Heuristically, propagation of atrribute data through call or return instructions does not change any attributes but only changes the interprocedural flow summary (call string) tagging each computed attribute.

We then define F^* to be the smallest set of functions acting in L^* which contains all the above functions H_I^*, $H^*_{(I,J)}$ and the identity map, and which is closed under functional composition and meet. For notational uniformity we will sometimes regard all the H* functions as edge functions, writing $H^*_{(I,J)}$ instead of H_I^* for all nonreturn instructions I and their successors J as well.

We will also extend the $\circ$ operation to apply to the interprocedural instruction-flow graph, which we denote as (N_I^*, E_I^*, I_0) , where N_I^* is the set of all instructions in the program being analyzed, where E_I^* is the set of all pairs of control-adjacent instructions (the transfer of control being either intraprocedural or interprocedural) and where I_0 is the entry instruction of the main program. We do this by defining for each $\gamma \in \Gamma$, $(I,J) \in E_I^*$

$$\gamma \circ (I,J) = \begin{cases} \gamma \circ (m,n), & \text{if I is a branch from node m to node n whose initial instruction is J} \\ \gamma, & \text{otherwise} \end{cases}$$

This extended definition allows us to extend the CM map and the notion of interprocedural validity of flow-graph paths to execution paths (i.e. paths in the instruction-flow graph) as well, and to define F^* in terms of the instruction flow graph alone.

Having defined (L^*,F^*) we can then apply the analysis technique outlined in Section 4 of [SP] to this framework (noting that (L^*,F^*) and (L,F) bear the relationship assumed there), to define an interprocedural maximal fixpoint solution $x^*: N^* \to L^*$. Likewise, we can also define a similar solution $z^*: N_I^* \to L^*$, which is the maximal fixpoint solution of the following set of equations (analogous to equations (3.1)):

$$(5.2) \qquad \begin{aligned} z^*_{I_0} &= \{(\lambda,\Omega^*)\}, \quad \text{where } \lambda \text{ is the null call string and } \Omega^* \text{ maps every } V \in \Sigma \text{ to } \Omega; \\ z^*_I &= \wedge \{H^*_{(J,I)}(z^*_J): (J,I) \in E^*_I\}, \quad \text{for all other instructions I.} \end{aligned}$$

Even though both the x* solution and the z* solution need not be effectively computable in the recursive case, they are both well defined, by the process described in Section 4 of [SP]. However, they are effectively computable if the program being analyzed contains no recursion.

We can now carry out an analysis quite analogous to that given in Section 3, which shows that the ud* and the flow-graph approaches to attribute flow analysis are also equivalent in the interprocedural case. We begin with

THEOREM 5.1. For each $n \in N^*$, $x^*_n = z^*_{I_n}$, where I_n is the initial instruction of n.

Proof: Completely analogous to the proof of Theorem 3.1, with the additional observation that for each $m \in N^*$, consisting of the instructions $J_1,\ldots,J_t$ and terminating at a branch to some other node $n \in N$, we have

$$f^*_{(m,n)} = H^*_{J_t} \circ H^*_{J_{t-1}} \circ \cdots \circ H^*_{J_1}$$

which can be informally interpreted as

$$(H_{J_t} \circ \cdots \circ H_{J_1})^* = H^*_{J_t} \circ \cdots \circ H^*_{J_1}$$

This is because call strings tagging data at J_1 do not change during the flow from J_1 to n (except possibly at the branch to n, which might be a procedure call or return). This observation allows us to formulate the proof of this theorem by replacing (L,F) by (L^*,F^*) in the proof of Theorem 3.1.

Q.E.D.

Next we have the following analog of Lemma 3.2, which shows how variable attributes at a given point depend on preceding definitions of that variable, and which ensures that only interprocedurally valid concatenations of execution paths are traced:

Lemma 5.2. For each $I \in N_I$, $V \in \Sigma$, $\gamma \in \Gamma$,

$$z_I^*(V)(\gamma) = \Lambda\{H_J^*(z_J^*)(V)(\gamma_1) : V \text{ is modidied in } J, \ \gamma_1 \in \Gamma \text{ and there exist paths } p_1 \in IVP(r_1,J) \cap CM^{-1}\{\gamma_1\}, \ p_0 \in path_{G^*}(J,I) \text{ such that } p_1 \| p_0 \in IVP(r_1,I) \cap CM^{-1}\{\gamma\} \text{ and } p_0 \text{ is free of modifications of } V\}. \tag{5.3}$$

Proof: First note that J cannot be an interprocedural jump, since we assume that such jumps do not modify any variables. The proof is again analogous to that of Lemma 3.2 and goes as follows. Denote the right-hand side of (5.3) by $\hat{z}_I^*(V)(\gamma)$. Let $I \in N_I$, $V \in \Sigma$, $\gamma \in \Gamma$ be given, and let $J \in N_I$ be an instruction appearing in the expression for $\hat{z}_I^*(V)(\gamma)$. Let $\gamma_1 \in \Gamma$, and let $p_0 = (J = K_1, K_2, \ldots K_s = I)$. and $p_1 \in IVP(r_1,J) \cap CM^{-1}\{\gamma_1\}$ be execution paths satisfying the condition in that expression. Then if we define

$$\gamma_2 = \gamma_1 \circ (J,K_2)$$
$$\gamma_3 = \gamma_2 \circ (K_2,K_3)$$
$$\vdots$$
$$\gamma_s = \gamma_{s-1} \circ (K_{s-1},I)$$

then it is easily seen that for all $2 \leq j \leq s$, γ_j is defined and $\gamma_s = \gamma$ (cf. e.g. Lemmas 3.1 and 4.1 of [SP]). By (5.2) we then have

$$
\begin{aligned}
z^*_{K_2}(V)(\gamma_2) &\leq H^*_{(J,K_2)}(z^*_J)(V)(\gamma_2) = H^*_J(z^*_J)(V)(\gamma_1) \\
z^*_{K_3}(V)(\gamma_3) &\leq H^*_{(K_2,K_3)}(z^*_{K_2})(V)(\gamma_3) \\
&= z^*_{K_2}(V)(\gamma_2) \\
&\vdots \\
z^*_I(V)(\gamma) &= z^*_I(V)(\gamma_s) \leq H^*_{(K_{s-1},I)}(z^*_{K_{s-1}})(V)(\gamma_s) \\
&= z^*_{K_{s-1}}(V)(\gamma_{s-1})
\end{aligned}
$$

Hence

$$z^*_I(V)(\gamma) \leq H^*_J(z^*_J)(V)(\gamma_1)$$

whence

$$z^*_I \leq \hat{z}^*_I$$

To prove the converse statement we show that if $z^{*(k)}_I$ denotes the value of z^*_I after k steps of the iterative process which defines that solution, and $\hat{z}^{*(k)}_I$ denotes the corresponding value of $\hat{z}^*_I$, then

(5.4) $z^{*(k)}_I(V)(\gamma) \geq \hat{z}^{*(k)}_I(V)(\gamma)$, for each $I \in N_I$, $V \in \Sigma, \gamma \in \Gamma$ and $k \geq 0$

which we prove by induction on k. As before, the case k = 0 is trivial, and let us assume for simplicity that each iteration step propagates information along a single edge in E^*_I. Assume that (5.4) is true for some $k \geq 0$, and let $(\tilde{J},\tilde{I})$ be the edge along which information is propagated at the (k+1)-st iteration. As before, (5.4) holds for k+1 for all $I \neq \tilde{I}$. If $I = \tilde{I}$, $V \in \Sigma$, $\gamma \in \Gamma$, then we have (assuming a propagation mechanism like that of Kildall [Ki])

$$z^{*(k+1)}_{\tilde{I}}(V)(\gamma) = z^{*(k)}_{\tilde{I}}(V)(\gamma) \wedge H^{*}_{(\tilde{J},\tilde{I})}(z^{*(k)}_{\tilde{I}})(V)(\gamma)$$

with no loss of generality, we may assume that $IVP(r_1,\tilde{I}) \cap CM^{-1}\{\gamma\} \neq \emptyset$, for otherwise $z^{*(k+1)}_{\tilde{I}}(V)(\gamma) = \Omega$ and (5.4) holds trivially. By the induction hypothesis, $z^{*(k)}_{\tilde{I}}(V)(\gamma) \geq \hat{z}^{*(k)}_{\tilde{I}}(V)(\gamma) \geq \hat{z}^{*(k+1)}_{\tilde{I}}(V)(\gamma)$. If V is modified in $\tilde{J}$, then $(\tilde{J},\tilde{I})$ is intraprocedural. It follows that $\tilde{J}$ appears in the expression for $\hat{z}^{*}_{I}(V)(\gamma)$ iff there exists a path $p \in CM^{-1}\{\gamma\} \cap IVP(r_1,\tilde{I})$ whose last edge is $(\tilde{J},\tilde{I})$ such that if we write $p = p_1 \| (\tilde{J},\tilde{I})$ then $CM(p_1) = \gamma$. If this is not the case, then $H^{*}_{(\tilde{J},\tilde{I})}(z^{*(k)}_{\tilde{J}})(V)(\gamma) = \Omega$, and (5.4) is again immediate. If this is the case, then, since $z^{*(k)} \geq z^{*(k+1)}$ and $H^{*}_{(\tilde{J},\tilde{I})} = H^{*}_{\tilde{J}}$ is monotone in L^{*}, we obtain

$$H^{*}_{(\tilde{J},\tilde{I})}(z^{*(k)}_{\tilde{J}})(V)(\gamma) \geq H^{*}_{\tilde{J}}(z^{*(k+1)}_{\tilde{J}})(V)(\gamma) \geq \hat{z}^{*(k+1)}_{\tilde{I}}(V)(\gamma)$$

so that (5.4) holds in this case as well. On the other hand, if V is not modified in $\tilde{J}$, let $\gamma_1 \in \Gamma$ such that $\gamma_1 \circ (\tilde{J},\tilde{I}) = \gamma$ (if no such γ_1 exists then the expression containing H^{*} is Ω and the desired inequality is immediate). Then, by the induction hypothesis

$$H^{*}_{(\tilde{J},\tilde{I})}(z^{*(k)}_{\tilde{J}})(V)(\gamma) = z^{*(k)}_{\tilde{J}}(V)(\gamma_1) \geq \hat{z}^{*(k)}_{\tilde{J}}(V)(\gamma_1) \geq \hat{z}^{*(k+1)}_{\tilde{J}}(V)(\gamma_1)$$

(Noting again that by our assumption, $CM^{-1}\{\gamma_1\} \cap IVP(r_1,\tilde{J}) \neq \emptyset$.) Let $K \in N_I$, $\gamma_2 \in \Gamma$ such that V is modified in K and there exist paths

$p_2 \in IVP(r_1,K) \cap CM^{-1}\{\gamma_2\}$, $p_0 \in path_{G^*}(K,\tilde{J})$ where $p_1 \equiv p_2\ p_0 \in IVP(r_1,\tilde{J}) \cap CM^{-1}\{\gamma_1\}$ and p_0 is free of modifications of V. Let $p = p_1{}^!(\tilde{J},\tilde{I}) = p_2{}^!(p_0\ (\tilde{J},\tilde{I}))$. Then $p_0{}^!(\tilde{J},\tilde{I})$ is also free of modifications of V and $p \in IVP(r_1,\tilde{I}) \cap CM^{-1}\{\gamma\}$ (which follows from Lemma 4.1 of [SP]). Hence

$$\hat{z}^{*(k+1)}_{\tilde{J}}(V)(\gamma_1) = \Lambda\ \{H^*_K(z^{*(k+1)}_K(V)(\gamma_2)\colon K,\ \gamma_2 \text{ as above}\}$$

$$\geq \hat{z}^{*(k+1)}_{\tilde{I}}(V)(\gamma)$$

because, by the argument given above, every K, γ_2 appearing in the meet above also appears in the expression for $\hat{z}^*_{\tilde{I}}(V)$. Hence we obtain (5.4) also for this case. Passing to the limit in k we obtain $z^*_{\tilde{I}}(V)(\gamma) \geq \hat{z}^*_{\tilde{I}}(V)(\gamma)$, from which the lemma readily follows. Q.E.D.

Continuing as in section 3, we next introduce a map $\widehat{attr}^*\colon \Pi \to L^*_0$, so that for each $I \in N^*_I$, $\gamma \in \Gamma$ and $V \in \Sigma$ which is an input arguemnt of I, we define

$$\widehat{attr}^*(V_I)(\gamma) = z^*_I(V)(\gamma)$$

and if V is an output variable of I, we define

$$\widehat{attr}^*(V_I)(\gamma) = H^*_I(z^*_I)(V)(\gamma)$$

Then we have

COROLLARY 5.3. For each $I \in N^*_I$, $\gamma \in \Gamma$ and each variable occurrence V_I in I we have

$$\widehat{attr}^*(V_I)(\gamma) \leq attr^*(V_I)(\gamma)\ .$$

Proof: As before, it suffices to show that $\widehat{attr}^*$ satisfies equations (5.1). Suppose first that V_I is an input occurrence in I. By (5.3) we have

$$\widehat{attr}^*(V_I)(\gamma) = z_I^*(V)(\gamma) = \Lambda\left\{H_J^*(z_J^*)(V)(\gamma_1): J, \gamma_1 \text{ as in (5.3)}\right\}$$

But by the definition of ud^*, J, γ_1 appear in the above expression iff $(V_J,\gamma_1) \in ud^*(V_I,\gamma)$. That is

$$\widehat{attr}^*(V_I)(\gamma) = \Lambda\{\widehat{attr}^*(V_J)(\gamma_1): (V_J,\gamma_1) \in ud^*(V_I,\gamma)\}$$

which is (5.1)(i). $\widehat{attr}^*$ also satisfies (5.1)(ii) which is immediate from its definition. Q.E.D.

LEMMA 5.4. For each $I \in N_I^*$, $\gamma \in \Gamma$ and each occurrence V_I in I we have

$$attr^*(V_I)(\gamma) = \widehat{attr}^*(V_I)(\gamma) .$$

Proof: It suffices to show that $attr^*(V_I)(\gamma) \leq \widehat{attr}^*(V_I)(\gamma)$. To show this, let $\widehat{attr}^{*(k)}(V_I)$ denote the value of this map after the first k iterations in the process defining the solution of (5.2).

We will show by induction on $k \geq 0$ that

$$\widehat{attr}^{*(k)}(V_I)(\gamma) \geq attr^*(V_I)(\gamma)$$

This, together with the continuity of the functions H^* (cf. Lemma 4.2 of [SP]) will imply the required inequality. It suffices to consider the case where $CM^{-1}\{\gamma\} \cap IVP(r_1,\tilde{I}) \neq \emptyset$. As before, the inequalities are trivial for k = 0. Assume

that they hold for some $k \geq 0$, and consider the (k+1)-st propagation step in the solution of (5.2), which propagates data along an edge $(\tilde{J},\tilde{I}) \in E^*_{\tilde{I}}$. For any $I \neq \tilde{I}$ the desired inequalities obviously also hold for k+1. If $I = \tilde{I}$ and $V_{\tilde{I}}$ is some input occurrence in $\tilde{I}$ then

$$\widehat{attr}^{*(k+1)}(V_{\tilde{I}})(\gamma) = z^{*(k+1)}_{\tilde{I}}(V)(\gamma) = z^{*(k)}_{\tilde{I}}(V)(\gamma) \wedge H^*_{(\tilde{J},\tilde{I})}(z^{*(k)}_{\tilde{J}})(V)(\gamma)$$

But

$$z^{*(k)}_{\tilde{I}}(V)(\gamma) = \widehat{attr}^{*(k)}(V_{\tilde{I}})(\gamma) \geq attr^*(V_{\tilde{I}})(\gamma)$$

by the induction hypothesis. If V is modified in J then

$$H^*_{(J,\tilde{I})}(z^{*(k)}_{\tilde{J}})(V)(\gamma) = \widehat{attr}^{*(k)}(V_{\tilde{J}})(\gamma) \geq attr^*(V_{\tilde{J}})(\gamma)$$

and $(V_{\tilde{J}},\gamma) \in ud^*(V_{\tilde{I}},\gamma)$, since $(\tilde{J},\tilde{I})$ is obviously intraprocedural. Hence, by (5.1) (i),

$$attr^*(V_{\tilde{J}})(\gamma) \geq attr^*(V_{\tilde{I}})(\gamma)$$

from which the required inequality readily follows.

On the other hand, if V is not modified in $\tilde{J}$, then by (5.4),

$$H^*_{(\tilde{J},\tilde{I})}(z^{*(k)}_{\tilde{J}})(V)(\gamma) = z^{*(k)}_{\tilde{J}}(V)(\gamma_1) \geq \hat{z}^{*(k)}_{\tilde{J}}(V)(\gamma_1) \geq \hat{z}^{*(k)}_{\tilde{I}}(V)(\gamma)$$

(if there exists γ_1 such that $\gamma_1 \circ (\tilde{J},\tilde{I}) = \gamma$; otherwise an obvious inequality holds), where the last inequality is obtained as in the proof of (5.4). As in Section 3, we obtain

$$\hat{z}^{*(k)}_{\tilde{I}}(V)(\gamma) = \Lambda\left\{\widehat{attr}^{*(k)}(V_K)(\gamma_2) : (V_K,\gamma_2) \in ud^*(V_{\tilde{I}},\gamma)\right\}$$

$$\geq \Lambda\left\{attr^*(V_K)(\gamma_2) : (V_K,\gamma_2) \in ud^*(V_{\tilde{I}},\gamma)\right\}$$

$$= attr^*(V_{\tilde{I}}) , \qquad \text{by (5.1) (i).}$$

so that in this case we also have $\widehat{attr}^{*(k+1)}(V_{\tilde{I}})(\gamma) \geq attr^*(V_{\tilde{I}})(\gamma)$. Finally, if $V_{\tilde{I}}$ is an output occurrence in $\tilde{I}$, the inequality is established using a completely analogous proof to the one given in Section 3. Q.E.D.

We can now summarize our analysis in the following theorem, which essentially states that the interprocedural ud^* approach yields the same solution as the interprocedural flow-graph approach.

THEOREM 5.5. Let $I \in N^*_I$ belong to a basic block $n \in N^*$ and be preceded in this block by $I_1,\ldots,I_s$.

(i) If V_I is an input occurrence in I then

$$attr^*(V_I)(\gamma) = H_{I_s} \circ H_{I_{s-1}} \circ \cdots \circ H_{I_1}\left(x^*_n(\gamma)\right)(V) \text{ ; for each } \gamma\in\Gamma.$$

(ii) If V_I is an output occurrence in I then

$$attr^*(V_I)(\gamma) = H_I \circ H_{I_s} \circ \cdots \circ H_{I_1}\left(x^*_n(\gamma)\right)(V) \quad \text{, for each } \gamma\in\Gamma.$$

Proof: Immediate, as in Section 3. (In addition we make use of the fact that control-flow within n is strictly intraprocedural.) Q.E.D.

Remarks.

(1) Note that, as in [SP], a final phase is usually required, which combines all data gathered for any variable occurrence into a single value, by computing

$$attr(V_I) = \bigwedge_{\gamma \in \Gamma} attr^*(V_I)(\gamma) \ .$$

Note, however, that in the nondistributive case $attr(V_I)$ need not be equal to $H_I \circ H_{I_s} \circ H_{I_{s-1}} \cdots \circ H_{I_1}(x'_n(V))$, where x'_n is as defined in formula (4.2) of [SP] (V_I being an output occurrence), but may constitute a better solution (simple examples using e.g. constant propagation with the classical counterexample to distributivity (cf. [He]) can be constructed to show strict inequality between these two solutions).

(2) The remark made at tne end of Section 3 still applies with obvious rewording, to the abstract interprocedural case discussed above.

We next extend and modify the methods devised in this section so far, to derive an interprocedural use-definition technique, which replaces the set Γ of all call strings by some finite approximation $\hat{\Gamma}$, so that it becomes a convergent implementable algorithm. Our treatment here is rather general, and concerns itself with establishing the equivalence between the use-definition approach and its flow-graph counterpart.

Our notations and assumptions in this section follow, though in rather simplified manner, those of Section 6 of [SP].

Let $\hat{\Gamma}, \omega, *$ and ECS be as defined there, where $\hat{\Gamma}$ denotes a semigroup of approximate call-strings, where ω and $*$ denote the identity element and binary operation for $\hat{\Gamma}$, and where ECS maps each procedure p in the program to the set of all approximate strings which can tag execution paths leading to the entry of p. We note that instead of the operation $\hat{\circ}$ we can define a map $R: E^* \to 2^{\hat{\Gamma}\times\hat{\Gamma}}$, so that for each $(m,n) \in E^*$, $R_{(m,n)}$ is a relation in $\hat{\Gamma}$, defined so that $\alpha_1 R_{(m,n)}\ \alpha_2$ iff $\alpha_2 \in \alpha_1 \hat{\circ} (m,n)$. Then the map $\widehat{CM}$ can simply be defined as

$$\widehat{CM}(q) = R_{(s_{k-1},s_k)}[R_{(s_{k-2},s_{k-1})} \cdots [R_{(s_1,s_2)}\{\omega\}] \cdots]$$

for each path $q = (s_1 = r_1, s_2, \ldots, s_k)$. The set $IAP(r_1,n)$, $n \in N^*$, can then be defined as

$$\{q \in path_{G^*}(r_1,n): \widehat{CM}(q) \neq \emptyset\}$$

Note that the data-flow framework can be constructed using only the relation-map R; in particular we have for each $(m,n) \in E^*$, $\xi \in L^*$, $\alpha \in \hat{\Gamma}$

$$f^*_{(m,n)}(\xi)(\alpha) = \Lambda \left\{ f_{(m,n)}(\xi(\alpha_1)): \alpha_1\ R_{(m,n)}\alpha \right\}$$

In addition, one needs to know the 'initial' element w of $\hat{\Gamma}$ (which tags the null information at the entry node) in order to define the associated data-flow problem (cf. (6.2) in [SP]). This observation enables us to view the approximative approach of Section 6 of [SP] as a special case of a generalized interprocedural framework, of which the abstract framework discussed in the previous section is also another special case,

realized by taking $\hat{\Gamma} = \Gamma$, $\omega = \lambda$ (the null call string), and defining, for each $(m,n) \in E^*$

$$R_{(m,n)} = \left\{(\gamma, \gamma \circ (m,n)) : \gamma \in \Gamma \mid \gamma \circ (m,n) \text{ is defined}\right\}$$

i.e. in this particular case $R_{(m,n)}$ is a partially defined function, so that $\widehat{CM}(q)$ is the singleton $\{CM(q)\}$ if $q \in IVP(r_1,n)$ for some $n \in N^*$, and is $\emptyset$ otherwise. These notations can be viewed as a special case of those used in the qualified data-flow analysis technique of Holley and Rosen [HR].

We therefore proceed to generalize the analysis performed earlier in this section to the case of an arbitrary choice of $\hat{\Gamma}$, ω and R, and show that in this more general case we also have equivalence of the flow-graph approach and the use-definition approach.

Not choosing to repeat the whole analysis once more, we only comment on the main modifications needed to carry out this generalization of the preceding analysis: Replace the map IVP by IAP, equalities of the form $CM(p) = \gamma$ by relations of the form $\gamma \in \widehat{CM}(p)$, Γ by $\hat{\Gamma}$, λ by ω. Then, in the definition of the H^* functions, define H^*_I, for each call instruction I, as

$$H^*_I(x^*)(\gamma) = \Lambda\left\{x^*(\gamma_1) : \gamma_1 \in \hat{\Gamma} \mid \gamma_1 \; R_{(m_I,r_p)} \; \gamma\right\}$$

where I calls p and m_I denotes the block consisting of I. Similar redefinition of the H^* map for procedure return jumps is used.

We assume that for each intraprocedural edge $(I,J) \in E_I^*$ $R_{(I,J)}$ is the identity relation on $\hat{\Gamma}$. Using this assumption, we can prove the appropriate variant of Theorem 5.1 in a completely analogous manner to the original proof.

In the first part of the modified proof of Lemma 5.2, we argue that it follows from the definition of CM that

there exist $\gamma_2, \gamma_3, \ldots, \gamma_s = \gamma$ such that

$\gamma_1 R_{(J,K_2)} \gamma_2$, $\gamma_2 R_{(K_2,K_3)} \gamma_3$, $\ldots$, $\gamma_{s-1} R_{(K_{s-1},I)} \gamma_s$.

In the chain of resulting inequalities we then have, e.g.

$$z^*_{K_3}(V)(\gamma_3) \leq H^*_{(K_2,K_3)}(z^*_{K_2})(V)(\gamma_3) = \wedge \left\{ z^*_{K_2}(V)(\gamma_3') : \gamma_3' \; R_{(K_2,K_3)} \; \gamma_3 \right\}$$

$$\leq z^*_{K_2}(V)(\gamma_2) \qquad \text{(since } \gamma_2 \; R_{(K_2,K_3)} \; \gamma_3\text{)}$$

etc.

Making similar modifications in subsequent proofs given earlier in this section, we can obtain an appropriate generalization of the preceding analysis which establishes the equivalence between our two attribute-flow analysis methods in the general case as well. Details are left to the reader.

The only remaining issue is how to compute the generalized ud* map, so that sharper interprocedural ud-based analysis can be performed. We now address ourselves to this problem.

A first possible technique, which is straightforward, but not very pragmatic, is to introduce a special data-flow problem which can be interpreted as an interprocedural exten-

sion of the classical 'reaching definitions' analysis (cf. [HE], e.g.). The ud^* map can then be derived from the solution of that analysis in much the same way as the standard ud map is derived from the solution of the reaching definition problem. This is done as follows.

Assume that $\hat{\Gamma}$ is chosen to be finite. Let $L = 2^{\Pi_D \times \hat{\Gamma}}$ where Π_D is the set of all (global) variable definitions/modifications in the program to be analyzed. More precisely, as will be seen below, re require elements of L to contain only pairs of the form (VO,γ), where $VO \in \Pi_D$ is a definition occurring at some procedure p of the program, and $\gamma \in ECS(p)$, i.e. γ can tag some execution path leading to VO from the program entry (here, with no loss of generality, we make an implicit assumption that all instructions within a procedure can be reached from its entry). Next, we define a new flow graph (N,E,r_0) where

$$N = N^* \times \hat{\Gamma};$$

$$E = \left\{(n_1\gamma_1,\ n_2\gamma_2):\ (n_1,n_2) \in E^* \text{ and } \gamma_1\ R_{(n_1,n_2)}\ \gamma_2\right\}$$

$$r_0 = (r_{main},\omega)$$

For each $(n_1\gamma_1,\ n_2\gamma_2) \in E$ and each $x \in L$ we define

$$f_{(n_1\gamma_1,n_2\gamma_2)}(x) = x \cap (nokill(n_1,n_2) \times \hat{\Gamma}) \cup (leave(n_1,n_2) \times \{\gamma_1\})$$

where $nokill(n_1,n_2)$ is the set of all variable definitions whose variables are not modified as control advances from n_1 to n_2, and $leave(n_1,n_2)$ is the set of all variable definitions within n_1 which can reach the start of n_2. We define

F as the smallest set of functions acting in L which contains all those functions and the identity and which is closed under functional compositions and meets. Obviously, F is distributive.

We can now associate the following data-flow problem with (L,F) and the new flow graph: Compute a minimal fixpoint solution $x: N \to L$ of the set of equations

$$x(r_0) = \emptyset \tag{5.5}$$
$$x(n\gamma) = \cup \left\{ f_{(n_1\gamma_1, n\gamma)}(x(n_1\gamma_1)) : (n_1\gamma_1, n\gamma) \in E \right\}$$

We have thus defined our data-flow problem using standard notation, so that we can apply standard techniques to solve it iteratively and obtain a solution which satisfies by Kildall's theorem [Ki]

$$x(n\gamma) = \cup \left\{ f_p(\emptyset) : p \in path_G(r_0, n\gamma) \right\}$$

The following can then easily be checked.

LEMMA 5.6. Let $p \in path_G(r_0, n\gamma)$. Write p as $\left((r_{main}, w), (n_2, \gamma_2), (n_3, \gamma_3), \ldots, (n_{s-1}, \gamma_{s-1}), (n, \gamma)\right)$. Then

(a) the path $\hat{p} = (r_{main}, n_2, n_3, \ldots, n) \in IAP(r_{main}, n)$ and $\gamma \in \widehat{CM}(\hat{p})$ (a similar statement obviously also holds for each initial subpath of $\hat{p}$)

(b) $(VO', \gamma') \in f_p(\emptyset)$ iff $\exists\, k < s$ such that $\gamma_k = \gamma'$ and $VO' \in leave(n_k, n_{k+1})$ and also $\in nckill(n_j, n_{j+1})$, $j = k+1, \ldots, s-1$.

COROLLARY 5.7. For each $(n,\gamma) \in N$, $x(n,\gamma)$ is the set of all pairs (VO',γ') such that VO' is a definition of some variable V for which there exist paths $p' \in IAP(r_1,VO') \cap \widehat{CM}^{-1}\{\gamma'\}$, $p_0 \in path_{G^*}(VO',n)$ such that $p'!p_0 \in IAP(r_1,n) \cap \widehat{CM}^{-1}\{\gamma\}$ and V is not redefined/modified along p_0.

The last corollary implies that the ud^* map can be computed from the solution x in a manner completely analogous to the way in which the intraprocedural ud map is computed from the solution to the reaching definitions analysis.

Having described this basic algorithm, we will now suggest some possible improvements.

First we observe that the R relations are nontrivial only for interprocedural edges. This means that we can break our analysis into an intraprocedural phase followed by a rather compact interprocedural phase, thereby gaining considerable efficiency. Specifically, we proceed as follows: At each procedure entry r_p and immediately after each procedure call c (a point denoted as c_+), we insert dummy *definitions* of each global variable V (denoted as V_{r_p} and V_{c_+} respectively). Similarly, before each procedure return e_p and just before each procedure call c (a point denoted as c_-), we insert dummy *uses* of each global variable V (denoted as V_{e_p} and V_{c_-} respectively). Having done this, we subject each procedure p to a standard intraprocedural

ud computation analysis, Let ud denote the union of all resulting maps (taken over all procedures in the program).

We next compute our ud^* map as follows: First introduce a map REACH so that for each global variable V, each instruction i which is either a procedure entry, an exit, a pre-call, or a post-call, and each $\gamma \in \hat{\Gamma}$,

$$\text{REACH}(i,\gamma,V) = \{(VO',\gamma') \in x(n_i,\gamma) \text{ such that } \text{var}(VO') = V\}$$

where n_i is the block containing i.

The following lemma can be shown.

LEMMA 5.8. The map REACH satisfies the following set of equations:

(i) For each procedure entry r_p , $\gamma \in \hat{\Gamma}$, V a global variable,

$$\text{REACH}(r_p,\gamma,V) = \cup\{\text{REACH}(c_-,\gamma_1,V): c_- \text{ is a call to } p, \gamma_1 \; R_{(c,r_p)} \; \gamma\}$$

(ii) For each post-call point c_+ , where c calls procedure p, $\gamma \in \hat{\Gamma}$ and V global,

(5.6)

$$\text{REACH}(c_+,\gamma,V) = \cup\{\text{REACH}(e_p,\gamma_1,V): \gamma_1 \; R_{(e_p,c_+)} \; \gamma\}$$

(iii) For each pre-call point or an exit J, which is in some procedure q, each $\gamma \in \text{ECS}\{q\}$ and a global V

$$\text{REACH}(J,\gamma,V) = \{(V_I,\gamma): V_I \in ud\{V_J\} \text{ and I is not a post-call or an entry (i.e. } V_I \text{ is not dummy)}$$

$$\cup \{\text{REACH}(I,\gamma,V): V_I \in ud\{V_J\} \text{ and I is a post-call or an entry}\}$$

Proof: (i) and (ii) are obvious from the equations defining the map x. (iii) is an easy consequence of Corollary 5.7, where we also note that the node (J,γ) is reachable from the entry node in our modified flow graph iff $\gamma \in ECS\{q\}$, according to the special nature of the R relations. Q.E.D.

We thus apply the equations of Lemma 5.8 iteratively to obtain a minimal fixpoint solution. It can also be checked that this solution indeed coincides with the definition of REACH in terms of the map x. Then the interprocedural ud* map can be computed as follows:

For each use $VO \in \Pi$ of a variable V, occurring in some procedure q, and each $\gamma \in ECS\{q\}$, we have

$$ud^*(VO,\gamma) = \{(VO',\gamma): VO' \in ud\{VO\} \text{ and is not dummy}\}$$
$$\cup \left\{REACH(J,\gamma,V): J \text{ is a post-call or the entry of } q \text{ and } V_J \in ud\{VO\}\right\}. \quad (5.7)$$

We omit here a proof of this formula.

To conclude, we propose a four-phase procedure to compute the ud* map:

(a) Perform intraprocedural computation of the ud map for each procedure in the program, annotated with dummy definitions and uses of relevant global variables as explained above.

(b) Compute the ECS map, using e.g. equations (6.1) of [SP].

(c) Compute the REACH map, by equations (5.6).

(d) Compute the ud^* map, by formula (5.7) given above.

This algorithm is rather appealing as it can be viewed as an extension of any existing intraprocedural ud computation algorithm. It is easy to check that the same algorithm, with some slight modifications can be applied for the computation of the BFROM map, referred to in an earlier comment.

REFERENCES

[AU] Aho, A. V. and Ullman, J. D., "Principles of Compiler Design," Addison-Wesley, 1977.

[Al_1] Allen, F. E., "Control-flow Analysis," Proc. Symp. Compiler Optimization, SIGPLAN Notices 5 (1970) 1-19.

[Al_2] Allen, F. E., "Interprocedural Data-Flow Analysis," Proc. IFIP (1974), North-Holland, 398-402.

[AC] Allen, F. E., and Cocke, J., "A Program Data-Flow Analysis Procedure," CACM 19 (1976) 137-147.

[Ba] Banning, J. P., "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliasing of Variables," Proc. 6th POPL Conf. (1979) 29-41.

[GW] Graham, S. L. and Wegman, M., "A Fast and Usually Linear Algorithm for Global Flow Analysis," JACM 23 (1976) 172-202.

[Ha] Harrison, W. H., "Compiler Analysis of the Value Ranges for Variables," IEEE Trans. on Software Engineering 3 (1977) 243-250.

[He] Hecht, M. S. "Flow Analysis of Computer Programs," Elsevier North-Holland, 1977.

[HR] Holley, H. and Rosen, B. K., "Qualified Data Flow Analysis," Proc. 7^{th} POPL Conference, 1980.

[HU] Hecht, M. S. and Ullman, J. D., "A Simple Algorithm for Global Data-Flow Analysis Problems," SIAM J. Computing 4 (1975) 519-532.

[JM_1] Jones, N. D. and Muchnick, S. S., "Binding Time Optimization in Programming Languages," Proc. 3^{rd} POPL Conf. (1976) 77-94.

[JM_2] Jones, N. D. and Muchnick, S.S., "Flow Analysis and Optimization of LISP-Like Structures," Proc. 6^{th} POPL Conf. (1979) 244-256.

[KU] Kam, J. B., and Ullman, J. D., "Global Data Flow Analysis

and Iterative Algorithms," JACM 23 (1976) 158-171.

[Ka] Kaplan, M. A., "Relational Data-Flow Analysis," Tech. Rep. #243, Dept. of E.E. & Comp. Sci. Princeton U 1978.

[KaU] Kaplan, M. A., and Ullman, J. D., "A General Scheme for the Automatic Inference of Variable Types," Proc. 5th POPL Conf. (1978) 60-75.

[Ki] Kildall, G. A., "A Unified Approach to Global Program Optimization," Proc. 1st POPL Conf. (1973) 194-206.

[MFS] Mintz, R., Fisher, G. A. and Sharir, M., "The Design of a Global Optimizer," Proc. SIGPLAN Conf. on Compiler Construction, August 1979.

[Ro_1] Rosen, B. K., "Monoids for Rapid Data-Flow Analysis," IBM Research Rep. RC-7032, Yorktown Heights, 1978.

[Ro_2] Rosen, B. K., "Data-Flow Analysis for Procedural Languages," JACM 26 (1979) 322-344.

[Sc_1] Schwartz, J. T., "Optimization of Very High-Level Languages, I, Value Transmission and Its Corollaries," J. Comp. Languages 1 (1975) 161-194.

[Sc_2] Schwartz, J. T., "Use-Use Chaining as a Technique in Typefinding," SETL Newsletter #140 (1974) Courant Inst. NYU.

[SS] Schwartz, J.T. and Sharir, M., "Design of Optimizations of the Bitvectoring Class", Courant Inst. Comp. Sci. Rep. #17, September 1979, New York University.

[SP] Sharir, M., and Pnueli, A., "Two Approaches to Interprocedural Data-Flow Analysis," Courant Inst. Comp. Sci. Tech. Rep. 2 (1978) (to appear in "Program Flow Analysis — Theory and Applications," Jones, N.D. and Muchnick, S.S. (Eds.) Prentice-Hall).

[Te] Tenenbaum, A.M., "Type Determination for Very High-Level Languages," Courant Comp. Sci. Rep. #3, Courant Inst. 1974.

NYU
Comp. Sci. Dept.
TR-017

c.1

Comp. Sci. Dept.
TR-017 Sharir

AUTHOR
ADA...

c.1

NYU
Comp. Sci. Dept.
TR-017 Sharir

AUTHOR
Application of the use-...

TITLE
definition chaining to ...

DATE DUE	BORROWER'S NAME

N.Y.U. Courant Institute of
Mathematical Sciences

This book may be kept

FOURTEEN DAYS

A fine will be charged for each day the book is kept overtime.
